A Type Inference System Based on Saturation of Subtyping Constraints

Benoît Vaugon, Michel Mauny

A Type Inference System Based on Saturation of Subtyping Constraints

Benoît Vaugon[1] and Michel Mauny[1,2]

[1] U2IS, ENSTA ParisTech, Université Paris-Saclay, 828 bd des Maréchaux, 91762 Palaiseau Cedex France

[2] INRIA Paris, 2 Rue Simone IFF, 75012 Paris France

Abstract. This paper[3] presents a powerful and flexible technique for defining type inference algorithms, on an ML-like language, that involve subtyping and whose soundness can be proved. We define a typing algorithm as a set of inference rules of three distinct forms: *typing* rules collect subtyping constraints to be satisfied, *instantiation* rules instantiate type schemes, and *saturation* rules specify how to check the validity and consistency of collected constraints. Essentially, type inference then proceeds in two intertwined phases: one that extracts constraints and the other that saturates the sets of constraints. Our technique extends easily to the treatment of high-level features such as polymorphism, overloading, variants and pattern-matching, or generalized algebraic data types (GADTs).

1 Introduction

The presentation of type synthesis of a program as the collection of constraints to be satisfied by its sub-expressions, followed by their resolution, is now classical. To cite only a few of them, [7, 12, 9] consider and solve equality constraints, and [14, 4] consider subtyping constraints.

In this work, we follow a "collect-and-saturate" approach in the spirit of [1] and [14], rather than "collect-and-solve". The main differences between our approach and [1, 14] is that we have a much simpler type language and a richer constraint language, providing in particular disjunctions[4] and negations of constraints. Moreover, we use a uniform formalism that allows us to both represent typing proofs and effectively implement a type inference algorithm. Finally, we do not try to generate *solutions* to our sets of constraints: we saturate them in order to check their consistency, an idea already present

[3] This work is part of the first author's PhD thesis [15].

[4] Disjunctions should not to be confused with union types.

in [1]. In contrast with constraint resolution, saturation allows us to continue to check compatibility of constraints for which finding a solution would become undecidable.

Our formalism is not really original: it is well known that syntax-directed inference rules can be used to define functions. Still, using them to define complete inference algorithms and to prove their soundness is original, as far as we know. The main contribution of this paper is therefore to develop a "collect-and-saturate" approach of subtyping constraints in a uniform framework, and show that it extends rather easily to high-level features of functional programming language.

This paper is organized as follows: section 2 presents the programming language that we consider, section 3 presents the type algebra and the constraint language, and section 4 introduces our base type inference system. Section 5 states the properties of this system and section 6 briefly describes its implementation. Finally, section 7 gives some hints about possible extensions of the type system and the appendix lists the full set of inference rules.

2 The language

The language that we consider here is given in figure 1: it is a functional language in the spirit of ML, with constants, primitive operations, data constructors (K_i) and pattern-matching with an optional default case.

$e ::= x \mid \lambda\ x\ .\ e \mid e_1\ e_2$	*λ-calculus*
$\mid c$	*constants*
$\mid (e_1, e_2)$	*pairs*
$\mid p^1\ e \mid p^2\ e_1\ e_2$	*primitive operations, including projections*
$\mid K\ e$	*data constructors*
$\mid$ `let` $x = e_1$ `in` e_2	*local declarations*
$\mid$ `if` e_1 `then` e_2 `else` e_3	*conditional*
$\mid$ `match` e `with` $K_1\ x_1 \to e_1 \parallel \ldots \parallel K_n\ x_n \to e_n$	*pattern-matching*
$\mid$ `match` e `with` $K_1\ x_1 \to e_1 \parallel \ldots \parallel K_n\ x_n \to e_n \parallel x_d \to e_d$	

Fig. 1. The expression language

The language has a classical call-by-value semantics[5].

[5] Note that the evaluation order does not interfere with typing, and laziness would not complicate the treatment that is given here.

3 Types and constraints

Types are distinguished according to whether they come from building values (we call them "left-types", that will occur at the left of subtyping constraints) or from value deconstruction ("right-types") such as pattern-matching or function application. Right-types include types of the form $\{\ \mathtt{K}_1\ \alpha_1 \parallel \ldots\ \}$ which correspond to deconstruction of variants by pattern-matching.

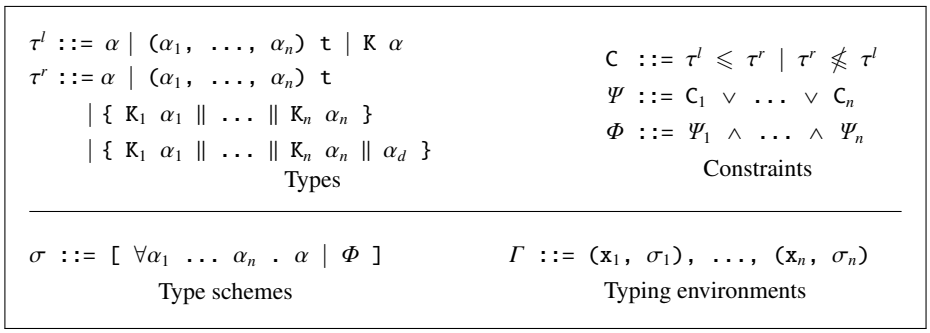

Fig. 2. Types and constraints

We use a single type construction ($(\alpha_1, ..., \alpha_n)\ \mathtt{t}$), with a postfix notation, to encode all native type constructors like `int`, `string`, etc. (for which $n = 0$), as well as product types ($(\alpha_1, \alpha_2)\ \times$) and arrow types ($(\alpha_1, \alpha_2)\ \rightarrow$). For the sake of readability, we use in the following the standard infix notations $(\alpha_1 \times \alpha_2)$ and $(\alpha_1 \rightarrow \alpha_2)$.

Using the same notation for all type constructors is possible since our saturation mechanism does not need any information about the variance of type parameters. Indeed, at saturation time, the type constructors $(\times)$ and $(\rightarrow)$ are always treated in the same way. Thanks to our typing rules, the initial orientation of constraints associated to a given type constructor is always sufficient to encode variance.

Note that our grammar of types is non-recursive. This property simplifies termination proofs, inference rules, and the encoding of type schemes. However, it remains possible to encode what we commonly call *recursive types* using cyclic dependencies in sets of constraints, like for example: $(\alpha \leqslant \beta \wedge \beta \leqslant \alpha \rightarrow \alpha)$. Enabling recursive subtyping constraints is not a problem for the inference mechanisms presented here, but could be confusing for the programmer since they would allow to write down code elements (functions, for instance) that will be typable but whose usage will be rejected in all

contexts. As usual, it is possible to forbid the production of recursive typing constraints by adding a verification on generated sets of constraints that detects cyclic dependencies between type variables.

Typing environments are equipped with two classical operations: one for adding a new binding $(\mathtt{x},\ \sigma)$ in a type environment Γ, written $\Gamma \oplus (\mathtt{x},\ \sigma)$, and the other to extract the type scheme associated to a variable $\mathtt{x}$ in Γ, written $\Gamma[\mathtt{x}]$.

Constraints can be direct ($\leqslant$) or negated ($\not\leqslant$), the latter occurring for instance when dealing with precise typing of pattern matching (not presented in this paper, see [15]). A conjunction Φ, which we sometimes call a *constraint set*, is made of disjunctions Ψ. When we write a disjunction as $\Psi \vee C$, we call Ψ the *alternative* to C.

Alternative constraints should not be confused with more common *disjunctive types* or *union types* that may be encoded with simple conjunctions between subtyping constraints. Disjunctions will only be useful for extensions to the language, introduced further. In particular, used with a negation, they allow to express *implications*. They also naturally appear when we negate a conjunction, as we need to implement GADTs with complete inference. The grammars of types and constraints are given in figure 2.

4 Inference systems

4.1 Inference rules

In our formalism, an inference system has three different kinds of rules, namely *typing* rules, that we sometimes call "T-rules", *instantiation* rules (I-rules), and *saturation* rules (S-rules).

Typing rules. The typing rules (whose names start with a "T") collect constraints. There is usually exactly one such rule per syntax construct. T-rules have the following shape:

$$\text{Txxx}\ \frac{\cdots \qquad \cdots \qquad \cdots}{\Phi, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha \rhd \Phi'}$$

Their conclusion should be read as "*under the set of constraints Φ, in the environment Γ: either e can be of type α, producing constraints which, when saturated with Φ, generate Φ'; or the disjunction Ψ is valid and compatible with Φ, and the saturation of $\Phi \wedge \Psi$ generates Φ'*". The set Φ' is the enrichment of Φ that is produced when the type synthesis of e has been performed. Of course, when $e : \alpha$, Φ' constraints α, and the "type" of e can be expressed as "α such that Φ'".

Saturation rules. Their names start with a "S" and they have the following shapes:

$$\text{Sxxx}\ \frac{\cdots \qquad \cdots \qquad \cdots}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'}$$

They should be read as: "*either $\tau^l \leqslant \tau^r$ is valid and compatible with Φ, or the disjunction Ψ is non-empty and compatible with Φ, Φ' being the resulting set of constraints*".

Instantiation rules. Similar to S-rules, I-rules have the shape:

$$\text{Ixxx}\ \frac{\cdots \qquad \cdots \qquad \cdots}{\Phi \vdash \Psi \vee \sigma \leqslant \tau^r \rhd \Phi'}$$

and compare a type scheme σ with a type.

In the following, we call "T-node" (resp. I-node, S-node) a node that is an instance of a T-rule (resp. I-rule, S-rule). We give, in appendix (section 8), the complete set of rules for the language given in figure 1.

Structure of inference trees. A successful type inference produces an inference tree composed of three successive layers. The lower layer (in green on figure 3) is built from typing rules, and is therefore isomorphic to the program. The middle layer, in red, is made of instantiation rules used when type schemes are generated for polymorphic constants (such as the empty list), polymorphic primitives (for instance, pair projections), and polymorphic constructs of the language (*e.g.* the `let` construct). Finally, the topmost layer is made of saturation rules (in blue).

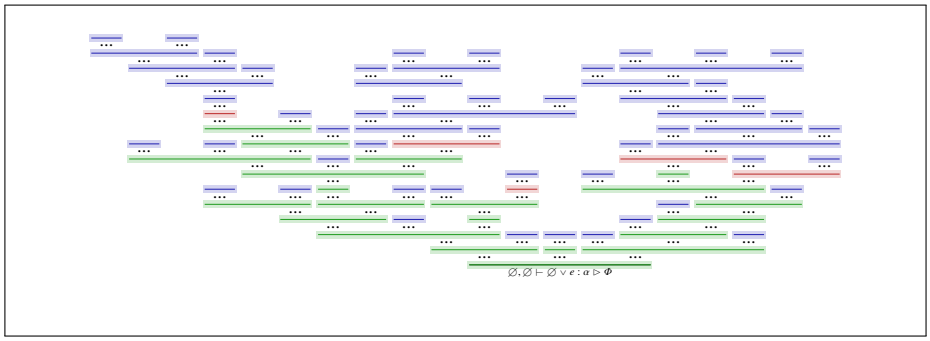

Fig. 3. The different layers of an inference tree

It is important to notice that the constraint set Φ *flows* through the tree. It is enriched with new constraints in the saturation subtrees, and used by saturation rules that check

compatibility of constraints, and by typing rules that generate type schemes at generalization time. We *only add* new constraints in Φ and there is no rule for "cleaning" it. In practice, optimizations are needed to remove subsets of *dead* or *redundant* constraints from Φ. They are not presented here, and can be found in Chap. 6 of [15].

4.2 Typability of an expression

Before giving examples of inference rules, and expressing the properties of our system, we need a few definitions and notations about type schemes.

Definition 1 (Order relation on type schemes). *Two type schemes* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ]$ *and* $[\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *satisfy* $[\ \forall\alpha_1 \dots \alpha_n\ .\ \alpha_0 \mid \Phi\ ] \leqslant [\ \forall\alpha'_1 \dots \alpha'_{n'}\ .\ \alpha'_0 \mid \Phi'\ ]$ *if there exists a substitution R of variables to variables (like a renaming, but not necessarily injective) such that:*

- $R(\alpha_0) = \alpha'_0$
- $R(\{\ \alpha_1, \dots, \alpha_n\ \}) \subset \{\ \alpha'_1, \dots, \alpha'_{n'}\ \}$
- $\forall\alpha\ .\ \alpha \notin \{\ \alpha_1, \dots, \alpha_n\ \} \Rightarrow R(\alpha) = \alpha$
- $R(\Phi) \subset \Phi'$

Intuitively, when we have $\sigma_1 \leqslant \sigma_2$, a value of type σ_1 can be used where a value of type σ_2 is expected. In other words, σ_1 is a subtype of σ_2.

We also need a generalization function, which, when given a typing environment, generalizes a type variable together with its constraints. Here is its definition:

$$\textsc{gen}(\alpha, \Phi, \Gamma) \triangleq [\ \forall(\textsc{ftv}(\Phi) \backslash \textsc{ftv}(\Gamma))\ .\ \alpha \mid \Phi\ ]$$

Given a particular inference system, we finally define a relation $(\_ : \_)$ that relates an expression e and a type scheme σ:

Definition 2 ($e : \sigma$). *We write $e : \sigma$ if, when given a type variable α, the two following properties hold:*

- *there exists Φ such that we have a proof tree of $\varnothing, \varnothing \vdash \varnothing \vee e : \alpha \rhd \Phi$*
- $\textsc{gen}(\alpha, \Phi, \varnothing) \leqslant \sigma$

We say that e is *typable* if there exists σ such that $e : \sigma$. Note that if $e : \sigma$, then $e : \sigma'$ for any σ' supertype of σ.

4.3 Examples of inference rules

The T-rule for a conditionalexpression ($\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$) generates a right-type `bool` (in deconstruction position) for the test e_1, and propagates the constraints from the two branches e_2 and e_3 to the result type α:

$$\textsc{TIf}\ \frac{\begin{array}{c}\texttt{let}\ \alpha'\ \textit{fresh} \\ \Phi_1 \vdash \Psi \vee \alpha' \leqslant \texttt{bool} \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash \Psi \vee e_1 : \alpha' \rhd \Phi_3 \\ \Phi_3, \Gamma \vdash \Psi \vee e_2 : \alpha \rhd \Phi_4 \qquad \Phi_4, \Gamma \vdash \Psi \vee e_3 : \alpha \rhd \Phi_5\end{array}}{\Phi_1, \Gamma \vdash \Psi \vee \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \alpha \rhd \Phi_5}$$

For most rules and in particular for TIF, the chaining of Φ's is arbitrarily chosen. In such cases, swapping children of rules does not change the set of typable programs. The order in which sub-expressions are analyzed and in which constraints are added in Φ is then chosen to follow the program source order, improving by the way the quality of error messages. The construction order of Φ is however constrained when a Φ is used by typing rules themselves, for example by the generalization mechanism applied on the `let` construction.

Abstractions build functional values, this is the reason why the T-rule for abstractions constraints the resulting type α to the left with an arrow type:

$$\textsc{TLambda}\ \frac{\begin{array}{c}\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \\ \Phi_1, \Gamma \oplus x : \alpha_1 \vdash \Psi \vee e : \alpha_2 \rhd \Phi_2 \qquad \Phi_2 \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi_3\end{array}}{\Phi_1, \Gamma \vdash \Psi \vee \lambda\, x\, .\, e : \alpha \rhd \Phi_3}$$

S-rules decompose constraints and saturate them by transitivity of subtyping. For this, we define auxiliary functions that extract components of constraint sets. As an example, the RIGHTS function extracts from Φ all right-types and alternatives Ψ associated to a given α:

$$\textsc{rights}(\alpha,\ \Phi) \ \triangleq\ \{\ (\Psi, \tau^r)\ |\ (\Psi \vee \alpha \leqslant \tau^r) \in \Phi\ \}$$

When adding a new disjunction $\Psi \vee \tau^l \leqslant \alpha$, we extract from Φ all right-types τ^r bigger than α, and their alternatives Ψ', and we generate disjunctions $\tau^l \leqslant \tau^r$, with $\Psi \vee \Psi'$ as alternative:

$$\textsc{STransRight}\ \frac{\begin{array}{c}\texttt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_n, \tau_n^r) = \textsc{rights}(\alpha,\ \Phi_1) \\ \Phi_1 \vdash \Psi \vee \Psi_1 \vee \tau^l \leqslant \tau_1^r \rhd \Phi_2 \quad \cdots \quad \Phi_n \vdash \Psi \vee \Psi_n \vee \tau^l \leqslant \tau_n^r \rhd \Phi_{n+1}\end{array}}{\Phi_1 \vdash \Psi \vee \tau^l \leqslant \alpha \rhd \Phi_{n+1}}$$

When a τ^l of the form K α, corresponding to the type inference of the application of a data constructor K, has to be compared to a $\{ \ldots, \mathrm{K}\ \alpha', \ldots \}$ coming from typing a pattern-matching that accepts values built with K, we propagate the subtype relation to type variables associated to the arguments of the data constructor:

$$\textsc{SVariantMatch}\ \frac{\Phi_1 \vdash \Psi \vee \alpha \leqslant \alpha' \rhd \Phi_2}{\Phi_1 \vdash \Psi \vee \mathrm{K}\ \alpha \leqslant \{ \ldots, \mathrm{K}\ \alpha', \ldots \} \rhd \Phi_2}$$

The saturation of constraints checks the validity and the compatibility of constraints (see the definitions below). When the validity of a constraint set cannot be checked by a saturation rule, the type inference fails and a type error is reported.

Definition 3 (Validity of a subtyping constraint C). *A subtyping constraint C is said to be* valid *if there exists a saturation rule whose conclusion has the form $\Phi \vdash \Psi \vee C \rhd \Phi'$, that is, if it is possible to perform at least one saturation step from $\Phi \vdash \Psi \vee C \rhd \Phi'$.*

In other words, a subtyping constraint is valid if it has one of the following forms:

- $\alpha \leqslant (\alpha'_1, \ldots, \alpha'_n)\ t$
- $(\alpha_1, \ldots, \alpha_n)\ t \leqslant \alpha'$
- $\alpha \leqslant \{ \ldots, K\ \alpha', \ldots \}$
- $K\ \alpha \leqslant \alpha'$
- $\alpha \leqslant \alpha'$
- $(\alpha_1, \ldots, \alpha_n)\ t \leqslant (\alpha'_1, \ldots, \alpha'_n)\ t$
- $K\ \alpha \leqslant \{ \ldots, K\ \alpha', \ldots \}$

Obviously, a disjunction of subtyping constraints Ψ is said to be valid if at least one of its members (a subtyping constraint) is valid. Similarly, a conjunction of subtyping relations Φ is valid if all its members are valid.

Definition 4 (Saturation of a constraint set Φ). *A set of constraints Φ is said to be* saturated *if it satisfies all the following properties:*

► *Comparison of parameterized types with the same name:*

- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha_1 \leqslant \alpha'_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha_n \leqslant \alpha'_n) \in \Phi \wedge$
  $(\Psi \vee \alpha'_1 \leqslant \alpha_1) \in \Phi \wedge \ldots \wedge (\Psi \vee \alpha'_n \leqslant \alpha_n) \in \Phi$

- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_n\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \nleqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t}) \in \Phi \Rightarrow$
  $(\Psi \vee \alpha_1 \nleqslant \alpha'_1 \vee \ldots \vee \alpha_n \nleqslant \alpha'_n \vee \alpha'_1 \nleqslant \alpha_1 \vee \ldots \vee \alpha'_n \nleqslant \alpha_n) \in \Phi$

► *When a disjunction member is invalid, the rest must be present in Φ:*

- $\forall\ \Psi, \alpha\ .\ (\Psi \vee \alpha \nleqslant \alpha) \in \Phi \Rightarrow \Psi \in \Phi$

- $\forall\ \Psi, \alpha_1, \ldots, \alpha_n, \mathtt{t}, \alpha'_1, \ldots, \alpha'_p, \mathtt{u}\ .$
  $(\Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{u}) \in \Phi \Rightarrow \Psi \in \Phi$

▶ *Transitivity through an α:*

- $\forall\ \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \alpha \leqslant \tau^r) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^l \leqslant \tau^r) \in \Phi$
- $\forall\ \Psi_1, \alpha, \tau^l, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \alpha \leqslant \tau^r) \in \Phi \wedge (\Psi_2 \vee \alpha \not\leqslant \tau^l) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \tau^l) \in \Phi$
- $\forall\ \Psi_1, \tau^l, \alpha, \Psi_2, \tau^r\ .$
  $(\Psi_1 \vee \tau^l \leqslant \alpha) \in \Phi \wedge (\Psi_2 \vee \tau^r \not\leqslant \alpha) \in \Phi \Rightarrow (\Psi_1 \vee \Psi_2 \vee \tau^r \not\leqslant \tau^l) \in \Phi$

The saturation of a set of constraints computes its smallest saturated superset.

4.4 Example

A complete example of the usage of these rules may be found in pages 46-47 of [15]. It details the inference tree obtained by type checking the expression (`not true`):

$$\frac{\quad\bigtriangledown\quad}{\varnothing, \varnothing \vdash \varnothing \vee \mathtt{not\ true} : \alpha \rhd \Phi}$$

As expected, the constructed Φ is, after cleaning, equivalent to:

$$\{\ \forall \alpha\ .\ \alpha \mid \mathtt{bool} \leqslant \alpha\ \}$$

5 Properties

Theorem 1 (Termination). *For a given finite expression e, the type inference of e (obtained by building the inference tree for $\varnothing, \varnothing \vdash \varnothing \vee e : \alpha \rhd \Phi$ following the given inference rules), always terminates.*

Since each inference rule builds exactly one node of the inference tree, proving that inference trees are always finite suffices to prove termination of the type inference.

Sketch of the proof. In order to prove that all inference trees have a finite size, we start by showing that the "typing part" of such a tree is finite since it is isomorphic to the syntax tree of the program, then that it contains only a finite number of I-nodes since they only appear at the frontier of the "typing part", and then that all S-subtrees have a finite size thanks to the structure of our types and constraints.

Theorem 2 (Soundness). *For any expression e and any type scheme σ such that $e : \sigma$, one of the following properties holds:*

- *evaluating e does not terminate,*
- *e evaluates to a value v and $v : \sigma$.*

The detailed proof of this theorem can be found in [15]. As in [5], we use a small-step semantics and a proof technique *à la Felleisen* (see [16]). The soundness theorem is a direct consequence of the two following lemmas:

- [Progress]: expressions that are not values and whose evaluation is blocked are untypable.
- [Subject Reduction]: if an expression e_1 is typable and evaluates to e_2 then e_2 is typable.

6 Implementation

The formalism that we have used for defining our inference systems can receive a direct implementation. Performing the type inference for an expression e only needs to build the inference tree by using the rules of the inference system under consideration.

```
let rec type_expr phi_1 env psi expr a =
  [...]
  | EIf (e1, e2, e3) ->
    let a' = Var.fresh() in
    let phi_2 = leq phi_1 psi (Var a') TBool  in
    let phi_3 = type_expr phi_2 env psi e1 a' in
    let phi_4 = type_expr phi_3 env psi e2 a  in
    let phi_5 = type_expr phi_4 env psi e3 a  in
    phi_5
  | ELambda (x, e) ->
    let a_1 = Var.fresh() and a_2 = Var.fresh() in
    let sub_env = (x, schema_of_var var1) :: env in
    let phi_2 = type_expr phi_1 sub_env psi e a_2 in
    let phi_3 = leq phi_2 psi (Arr a_1 a_2) (Var a) in
    phi_3
  [...]
```

Fig. 4. Systematic Implementation of Typing Rules TIF and TLAMBDA

At each step of building the tree, either no rule can be applied, and the type inference stops, raising an error, or there is exactly one rule that applies, and type inference goes on, deterministically. This property is true for the "typing part", as well as for "instantiation part" and the "saturation part". Furthermore, each rule precisely mentions the type variables to be generated, and there is no (implicit) global operation to perform, such as, for instance, renaming of type variables.

It is therefore extremely easy to extract a working implementation of an inference algorithm from such a type inference system. Given an expression e as input, one only has to generate a fresh type variable α and try to derive the inference tree from the root: $\dfrac{}{\varnothing, \varnothing \vdash \varnothing \vee e : \alpha \rhd \Phi}$. If the construction of this tree succeeds, it generates a set of constraints Φ and the inferred type for e is $\textsc{gen}(\alpha, \Phi, \varnothing)$.

Of course, it is not necessary to keep the whole inference tree in memory and a simple recursive algorithm is sufficient to extract constraints, check their compatibility and compute Φ. Such an implementation consists in three recursive functions, one for each kind of inference rule (typing, instantiation and saturation) in which each rule is implemented as a case. Figure 4 shows how the *typing* function may be systematically be derived from *typing* rules.

Of course, proceeding this way results in an implementation much less efficient than unification-based systems that use mutable data structures to represent type variables, and the famous union-find algorithm (*cf.* [3, 6]). Our systems simply cannot use unification, and we cannot use those tools directly to deal with our subtyping constraints.
The fact is that such a direct implementation of our systems produces an extremely inefficient resulting type inference program. The reason is threefold:

1. disjunctions are problematic: they are the source of combinatorial explosions in the saturation mechanisms;
2. the generalization mechanism naïvely encapsulates the whole set of constraints and provokes an explosion of the number of type variables generated at each instantiation, as well as an explosion of the number of α-renamed constraints;
3. the saturation mechanism keeps in Φ constraints that are consequences of others and generates an explosion of the number of items returned by $\textsc{lefts}$ and $\textsc{rights}$, resulting in useless constraints and computations.

Another difficulty with this naïve implementation technique lies in the readability of the type schemes provided to the user: their size becomes quickly huge (for the reasons above), and a simple clean up using dependency analysis is practically insufficient for having readable results.

However, we found two effective ways to improve the performance of our type inference prototype:

- clean up the sets of constraints contained in type schemes. In particular, we implemented a dependency analysis in the spirit of [10]. However, since the main performance issues in our systems come from disjunctions, we need to go further and develop other orthogonal techniques:
  - an improved dependency analysis using weak dependencies between several disjunctions of constraints;
  - a reinforcement of saturation, considering all disjunctions "modulo rotation".

  Furthermore, Pottier in [10] removes constraints using a detection of "equivalent subsets of constraints". We use the same idea but with a more aggressive definition of "equivalence". Rather than simply detecting isomorphic sub-graphs of constraints, we use the definition of *a set of constraint is less general than an other one*, originally designed to typecheck polymorphic recursion.
- use a dedicated representation of constraint sets containing disjunctions: this representation allows for optimizing the primitives that are used by cleaning algorithms, and limits some redundant computations during saturation.

These techniques have been tested and seem to work well in practice. They bring gains of several orders of magnitude in computing time as well as in the size of constraint sets. The cleaning mechanisms provide us with a further advantage: they also improve the readability of type schemes.

7 Extensions

The main advantages of the inference mechanism presented in this paper are its flexibility and its extensibility. Indeed, the first author's PhD thesis [15] extends the basic type system presented here in three orthogonal directions. We simply show here some hints about how these extensions work.

7.1 A finer typing of pattern matching on variants

The goal of this extension is to accept codes like the one given at figure 5, where the types of parameters x and y depend on the actual value of kind. This problem has been already studied elsewhere: some authors define a dedicated form of *implication*, *e.g.* by extending the grammar of types, defining *conditional types* [2, 8] and others extend the grammar of constraints, defining *conditional constraints* [11].

```
let sum x y kind = match kind with
  | INTEGERS -> x + y
  | STRINGS  -> concat x y in
let n = 2 * (sum 3 4 INTEGERS) in
let s = concat ">␣" (sum "Hello␣" "world" STRINGS) in
[...]
```

Fig. 5. An example of matching with different return types

Despite the fact that it is more intuitive to extend the grammar of types with expressions saying that "if the argument is of type X then the result is of type Y", this technique fails to link the type of an argument with arbitrary other types, *i.e.* types that are not directly related to this argument like, for example, the type of a previous argument or the type of a variable from an outer function.

The work of Pottier [11], using conditions at the level of constraints, avoids this problem and is the closest to ours. The main difference is that we do not design here a dedicated implication mechanism but instead use negations and disjunctions provided by our general framework.

The idea here consists in simply modifying the typing mechanism of pattern matching by associating in Φ, for each case, the variant being matched to the subtyping constraints extracted from the body of the case. This way, if a typing error occurs later because of a constraint coming from a case, instead of raising a type error immediately, we just mark that case as impossible.

It appears that our language of constraints is sufficient to express this kind of relation between variants and other arbitrary constraints. For each case, we simply use a negation to encode the fact that the set of values denoted by the type variable associated to the matched expression does not contain the variant from the current pattern, and disjunctions to link this constraint with those obtained when typing the body of the case. The saturation mechanism presented in this paper finishes the job.

7.2 A new generalization mechanism

The extension of pattern matching presented above is nearly sufficient to encode objects by message passing and without any extension of the language. In this setting, an *object*

is simply a function taking a method name (encoded as a variant) as its first argument and performing pattern matching to jump to the code associated to that method.

Unfortunately, the underlying type system is not powerful enough to accept codes like the following one, rather classical in object oriented languages like OCaml:

```
let f obj =
  println (obj ToString);
  obj GetWidth + 10 in
let mkobj width meth =
  match meth with
  | ToString -> concat "width:" (string_of_int width)
  | GetWidth -> width in
let obj = mkobj 42 in
f obj
```

Indeed, the `obj` parameter of `f` is used multiple times in different typing contexts, and the saturation leads to a clash on multiple occurrences of the unique type variable associated to `obj`.

The first motivation for our extension of polymorphism is to accept this kind of code. Many papers have been dedicated to extend the ML polymorphism, defining in particular different extensions and restrictions of the k-CFA, as described in [13].

The basic idea of our extension consists in delaying instantiation by extending the language of left-types (τ^l) with schemes (σ) and changing the instantiation rule into a saturation one performing instantiation as late as possible.

The recurrent problem with this kind of extension is that it breaks the termination of type inference. The originality of our approach concerns the technique that we use to ensure the termination by limiting the generalization power of our system, technique that can be understood by the programmer. Indeed, our technique simply limits the depth of nested polymorphic instantiation through function parameters in a context-free way. As we can see in [15], this technique is, once more, based on an extension of language of constraints.

7.3 GADTs with complete inference

In our context, we encode GADTs by:

- Adding a construction to the language to declare GADTs with local typing constraints.
- Adapting pattern matching on variants to GADTs by inserting in Ψ the negation of constraints specified by the user, like in our first extension (section 7.1).
- Extending the type language with a new construction for existential types.

The problem that usually breaks inference in the presence of GADTs is the management of existential types. In our system, this problem appears in saturation rules that take constraints containing existential types in their conclusion. Thanks to the presence of disjunctions and negations in the language of constraints, our saturation mechanism is able to automatically propagate constraints on existential types to other types occurring in the definition of the GADTs.

Each of these extensions makes intensive usage of the saturation mechanism, and the first and third ones perform heavy use of negations and disjunctions. This highlights the interest of a general framework like ours that factorizes formalization, implementation and optimization of a saturation mechanism on constraints using all operators from first order logic.

8 Conclusion

We have presented in this paper a type inference system based of saturation of subtyping constraints, providing terminating and sound inference algorithms. This system has been successfully extended to perform type inference of the following language features:

- overloading (with dynamic dispatch),
- more precise typing of pattern matching (that keeps the relationship between input patterns and output results),
- a generalization of ML polymorphism that enables polymorphic usages of function arguments,
- type inference for GADTs with subtyping.

We gave hints on how these extensions work. All but overloading are fully described in the first author's PhD thesis [15].

References

[1] Alexander Aiken and Edward L. Wimmers. "Type Inclusion Constraints and Type Inference". In: *Functional Programming and Computer Architecture*. Copenhagen, Denmark: ACM, 1993, pp. 31–41.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. “Soft Typing with Conditional Types”. In: *Principles of Programming Languages*. Portland, Oregon, USA: ACM, 1994, pp. 163–173.

[3] Arthur Charguéraud and François Pottier. “Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation”. In: *Interactive Theorem Proving*. Vol. 9236. Aug. 2015, pp. 137–153.

[4] Stephen Dolan and Alan Mycroft. “Polymorphism, subtyping and type inference in MLsub”. In: *Principles of Programming Languages*. 2017.

[5] Jonathan Eifrig, Scott Smith, and Valery Trifonov. “Type Inference for Recursively Constrained Types and its Application to OOP”. In: *Electronic Notes in Theoretical Computer Science* 1 (1995), pp. 132 –153.

[6] Fritz Henglein. “Efficient Type Inference for Higher-Order Binding-Time Analysis”. In: *Functional Programming and Computer Architecture*. 1991, pp. 448–472.

[7] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: Transactions on Programming Languages and Systems 4.2 (Apr. 1982), pp. 258–282.

[8] Zachary Palmer, Pottayil Harisanker Menon, Alexander Rozenshteyn, and Scott Smith. “Types for Flexible Objects”. In: *Asian Symposium on Programming Languages and Systems*. 2014, pp. 99–119.

[9] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. “Simple Unification-based Type Inference for GADTs”. In: SIGPLAN Notices 41.9 (Sept. 2006), pp. 50–61.

[10] François Pottier. “A Framework for Type Inference with Subtyping”. In: *International Conference on Functional Programming*. Sept. 1998, pp. 228–238.

[11] François Pottier. “A Versatile Constraint-Based Type Inference System”. In: Nordic Journal of Computing 7.4 (Nov. 2000), pp. 312–347.

[12] Vincent Simonet and François Pottier. *Constraint-Based Type Inference for Guarded Algebraic Data Types*. Research Report 5462. INRIA, Jan. 2005.

[13] Scott F. Smith and Tiejun Wang. “Polyvariant Flow Analysis with Constrained Types”. In: *European Symposium on Programming*. 2000, pp. 382–396.

[14] Valery Trifonov and Scott Smith. “Subtyping Constrained Types”. In: *Static Analysis Symposium*. 1996, pp. 349–365.

[15] Benoît Vaugon. “Sous-typage par saturation de contraintes: théorie et implémentation”. PhD thesis. Université Paris-Saclay, 2016. URL: `https://pastel.archives-ouvertes.fr/tel-01356695/document`.

[16] Andrew K. Wright and Matthias Felleisen. “A Syntactic Approach to Type Soundness”. In: Information & Computation 115 (1992), pp. 38–94.

Appendix

We give here the complete set of rules for the language given in figure 1.

Meta-functions

$$\begin{aligned}
\textsc{lefts}(\alpha,\ \Phi) &\triangleq \{\ (\Psi, \tau^l) \mid (\Psi \vee \tau^l \leqslant \alpha) \in \Phi\ \} \\
\textsc{rights}(\alpha,\ \Phi) &\triangleq \{\ (\Psi, \tau^r) \mid (\Psi \vee \alpha \leqslant \tau^r) \in \Phi\ \} \\
\overline{\textsc{lefts}}(\alpha,\ \Phi) &\triangleq \{\ (\Psi, \tau^r) \mid (\Psi \vee \tau^r \not\leqslant \alpha) \in \Phi\ \} \\
\overline{\textsc{rights}}(\alpha,\ \Phi) &\triangleq \{\ (\Psi, \tau^l) \mid (\Psi \vee \alpha \not\leqslant \tau^l) \in \Phi\ \}
\end{aligned}$$

The T meta-function associates a type scheme to constants and primitives. For instance:

- $T(3) \triangleq$ `[` $\forall\alpha$ `.` α `|` `int` $\leqslant \alpha$ `]`
- $T(\texttt{not}) \triangleq$ `[` $\forall\alpha\alpha_1\alpha_2$ `.` α `|` $\alpha_1 \to \alpha_2 \leqslant \alpha \ \wedge\ \alpha_1 \leqslant$ `bool` $\wedge$ `bool` $\leqslant \alpha_2$ `]`

Typing

$$\textsc{TConst}\ \frac{\Phi \vdash \Psi \vee T(\mathtt{c}) \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{c} : \alpha \rhd \Phi'} \qquad \textsc{TVar}\ \frac{\texttt{when}\ \Gamma[\mathtt{x}]\ \textit{defined} \qquad \Phi \vdash \Psi \vee \Gamma[\mathtt{x}] \leqslant \alpha \rhd \Phi'}{\Phi, \Gamma \vdash \Psi \vee \mathtt{x} : \alpha \rhd \Phi'}$$

$$\textsc{TApplyPrim1}\ \frac{\begin{array}{l}\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi \vdash \Psi \vee T(\mathtt{p}^1) \leqslant \alpha_1 \to \alpha_2 \rhd \Phi' \\ \Phi' \vdash \Psi \vee \alpha_2 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^1\ \mathtt{e}_1 : \alpha \rhd \Phi'''}$$

$$\textsc{TApplyPrim2}\ \frac{\begin{array}{l}\texttt{let}\ \alpha_0, \alpha_1, \alpha_2, \alpha_3\ \textit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_0 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \\ \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee T(\mathtt{p}^2) \leqslant \alpha_1 \to \alpha_0 \rhd \Phi''' \\ \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi'''' \qquad \Phi'''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi'''''\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{p}^2\ \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi'''''}$$

$$\textsc{TLambda}\ \frac{\texttt{let}\ \alpha_1, \alpha_2\ \textit{fresh} \qquad \Phi, \Gamma \oplus (\mathtt{x}, \alpha_1) \vdash \Psi \vee \mathtt{e} : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_1 \to \alpha_2 \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \lambda\,\mathtt{x}\,.\,\mathtt{e} : \alpha \rhd \Phi''}$$

TApp

$$\frac{\texttt{let}\ \alpha_1, \alpha_2, \alpha_3\ \mathit{fresh} \qquad \Phi \vdash \Psi \vee \alpha_1 \leqslant \alpha_2 \to \alpha_3 \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \alpha_3 \leqslant \alpha \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi''''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1\ \mathtt{e}_2 : \alpha \rhd \Phi''''}$$

TPair

$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \mathit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha_1 \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha_2 \rhd \Phi'' \qquad \Phi'' \vdash \Psi \vee \alpha_1 \times \alpha_2 \leqslant \alpha \rhd \Phi'''}{\Phi, \Gamma \vdash \Psi \vee (\mathtt{e}_1,\ \mathtt{e}_2) : \alpha \rhd \Phi'''}$$

TConstr

$$\frac{\texttt{let}\ \alpha'\ \mathit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha' \rhd \Phi' \qquad \Phi' \vdash \Psi \vee \mathtt{K}\ \alpha' \leqslant \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{K}\ \mathtt{e} : \alpha \rhd \Phi''}$$

TIf

$$\frac{\texttt{let}\ \alpha'\ \mathit{fresh} \qquad \Phi \vdash \Psi \vee \alpha' \leqslant \mathtt{bool} \rhd \Phi' \qquad \Phi', \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha' \rhd \Phi'' \qquad \Phi'', \Gamma \vdash \Psi \vee \mathtt{e}_2 : \alpha \rhd \Phi''' \qquad \Phi''', \Gamma \vdash \Psi \vee \mathtt{e}_3 : \alpha \rhd \Phi''''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \alpha \rhd \Phi''''}$$

TLet

$$\frac{\texttt{let}\ \alpha'\ \mathit{fresh} \qquad \Phi, \Gamma \vdash \Psi \vee \mathtt{e}_1 : \alpha' \rhd \Phi' \qquad \Phi', \Gamma \oplus (\mathtt{x}, \textsc{gen}(\alpha', \Phi', \Gamma)) \vdash \Psi \vee \mathtt{e}_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash \Psi \vee \mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \alpha \rhd \Phi''}$$

TMatch

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n\ \mathit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\mathtt{x}_1,\ \alpha_1) \vdash \Psi \vee \mathtt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\mathtt{x}_n,\ \alpha_n) \vdash \Psi \vee \mathtt{e}_n : \alpha \rhd \Phi_{n+1}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n : \alpha \rhd \Phi_{n+1}}$$

TMatchDefault

$$\frac{\begin{array}{c}\texttt{let}\ \alpha_e, \alpha_1, \ldots, \alpha_n, \alpha_d\ \mathit{fresh} \\ \Phi \vdash \Psi \vee \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \parallel \alpha_d \,\} \rhd \Phi_0 \qquad \Phi_0, \Gamma \vdash \Psi \vee \mathtt{e} : \alpha_e \rhd \Phi_1 \\ \Phi_1, \Gamma \oplus (\mathtt{x}_1,\ \alpha_1) \vdash \Psi \vee \mathtt{e}_1 : \alpha \rhd \Phi_2 \\ \ldots \\ \Phi_n, \Gamma \oplus (\mathtt{x}_n,\ \alpha_n) \vdash \Psi \vee \mathtt{e}_n : \alpha \rhd \Phi_{n+1} \\ \Phi_{n+1}, \Gamma \oplus (\mathtt{x}_d,\ \alpha_d) \vdash \Psi \vee \alpha_e \leqslant \{\, \mathtt{K}_1\ \alpha_1 \parallel \ldots \parallel \mathtt{K}_n\ \alpha_n \,\} \vee \mathtt{e}_d : \alpha \rhd \Phi_{n+2}\end{array}}{\Phi, \Gamma \vdash \Psi \vee \mathtt{match}\ \mathtt{e}\ \mathtt{with}\ \mathtt{K}_1\ \mathtt{x}_1 \to \mathtt{e}_1 \parallel \ldots \parallel \mathtt{K}_n\ \mathtt{x}_n \to \mathtt{e}_n \parallel \mathtt{x}_d \to \mathtt{e}_d : \alpha \rhd \Phi_{n+2}}$$

Instantiation

INST

$$\frac{\texttt{let } \alpha'_1, \ldots, \alpha'_n \textit{ fresh} \quad \Phi \vdash \Psi \vee \alpha_0[\alpha_i \mapsto \alpha'_i]_{i=1}^n \leqslant \tau^r \rhd \Phi_1 \quad \Phi_1 \vdash \Psi \vee \Psi_1[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_2 \quad \cdots \quad \Phi_p \vdash \Psi \vee \Psi_p[\alpha_i \mapsto \alpha'_i]_{i=1}^n \rhd \Phi_{p+1}}{\Phi \vdash \Psi \vee [\, \forall \alpha_1 \ \ldots \ \alpha_n \,.\, \alpha_0 \mid \Psi_1 \wedge \cdots \wedge \Psi_p \,] \leqslant \tau^r \rhd \Phi_{p+1}}$$

Saturation

To prevent our algorithm from entering in an infinite loop in the presence of cyclic relations between type variables, before checking the compatibility of a constraint with Φ, we check if this constraint is already present in Φ. If not, one of SNEWCONSTRAINT($\leqslant$) or SNEWCONSTRAINT($\not\leqslant$) is used, and we add the new constraint in Φ and generate a property annotated with a question mark ($\stackrel{?}{\vdash}$) consumed by rules defined further. If so, one of the axioms SALREADYPROVED($\leqslant$) or SALREADYPROVED($\not\leqslant$) is used and no more constraint is generated.

SNEWCONSTRAINT($\leqslant$)

$$\frac{\texttt{when } (\Psi \vee \tau^l \leqslant \tau^r) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \leqslant \tau^r) \stackrel{?}{\vdash} \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi'}$$

SNEWCONSTRAINT($\not\leqslant$)

$$\frac{\texttt{when } (\Psi \vee \tau^l \not\leqslant \tau^r) \notin \Phi \qquad \Phi \wedge (\Psi \vee \tau^l \not\leqslant \tau^r) \stackrel{?}{\vdash} \Psi \vee \tau^l \not\leqslant \tau^r \rhd \Phi'}{\Phi \vdash \Psi \vee \tau^l \not\leqslant \tau^r \rhd \Phi'}$$

SALREADYPROVED($\leqslant$)

$$\frac{\texttt{when } (\Psi \vee \tau^l \leqslant \tau^r) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \leqslant \tau^r \rhd \Phi}$$

SALREADYPROVED($\not\leqslant$)

$$\frac{\texttt{when } (\Psi \vee \tau^l \not\leqslant \tau^r) \in \Phi}{\Phi \vdash \Psi \vee \tau^l \not\leqslant \tau^r \rhd \Phi}$$

SLEQSAMEVAR

$$\frac{}{\Phi \stackrel{?}{\vdash} \Psi \vee \alpha \leqslant \alpha \rhd \Phi}$$

SNOTLEQSAMEVAR

$$\frac{\Phi \vdash \Psi \vee \phi \rhd \Phi'}{\Phi \stackrel{?}{\vdash} \Psi \vee \phi \vee \alpha \not\leqslant \alpha \rhd \Phi'}$$

SLEQSAMEPARAMED

$$\frac{\Phi \vdash \{\ \Psi \vee \alpha_i \leqslant \alpha'_i\ \}_{i=1}^n \rhd \Phi' \qquad \Phi' \vdash \{\ \Psi \vee \alpha'_i \leqslant \alpha_i\ \}_{i=1}^n \rhd \Phi''}{\Phi \stackrel{?}{\vdash} \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leq (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t} \rhd \Phi''}$$

SNOTLEQSAMEPARAMED

$$\frac{\Phi \vdash \Psi \ \vee\ \alpha_1 \not\leqslant \alpha'_1 \vee \cdots \vee \alpha_n \not\leqslant \alpha'_n \ \vee\ \alpha'_1 \not\leqslant \alpha_1 \vee \cdots \vee \alpha'_n \not\leqslant \alpha_n \rhd \Phi'}{\Phi \stackrel{?}{\vdash} \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant (\alpha'_1, \ldots, \alpha'_n)\ \mathtt{t} \rhd \Phi'}$$

SLeqDiffParamed

$$\frac{\Phi \vdash \Psi \vee \phi \rhd \Phi'}{\Phi \vdash^{?} \Psi \vee \phi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leq (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{u} \rhd \Phi'}$$

SNotLeqDiffParamed

$$\frac{\mathtt{when}\ \mathtt{t} \neq \mathtt{u}}{\Phi \vdash^{?} \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leq (\alpha'_1, \ldots, \alpha'_p)\ \mathtt{u} \rhd \Phi}$$

SVarLeqParamed

$$\frac{\begin{array}{c}\mathtt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_p, \tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) \qquad \mathtt{let}\ (\Psi'_1, \tau_1'^l), \ldots, (\Psi'_q, \tau_q'^l) = \overline{\textsc{rights}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^l \leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}\ \}_{i=1}^{p} \rhd \Phi' \\ \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \tau_i'^l\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash^{?} \Psi \vee \alpha \leq (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi''}$$

SParamedLeqVar

$$\frac{\begin{array}{c}\mathtt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_p, \tau_p^r) = \textsc{rights}(\alpha,\ \Phi) \qquad \mathtt{let}\ (\Psi'_1, \tau_1'^r), \ldots, (\Psi'_q, \tau_q'^r) = \overline{\textsc{lefts}}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leqslant \tau_i^r\ \}_{i=1}^{p} \rhd \Phi' \\ \Phi' \vdash \{\ \Psi \vee \Psi'_i \vee \tau_i'^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}\ \}_{i=1}^{q} \rhd \Phi''\end{array}}{\Phi \vdash^{?} \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \leq \alpha \rhd \Phi''}$$

SVarLeqVar

$$\frac{\begin{array}{c}\mathtt{when}\ \alpha_1 \neq \alpha_2 \\ \mathtt{let}\ (\Psi_1^l, \tau_1^l), \ldots, (\Psi_p^l, \tau_p^l) = \textsc{lefts}(\alpha_1,\ \Phi),\ (\varnothing, \alpha_1) \qquad \mathtt{let}\ (\Psi_1'^l, \tau_1'^l), \ldots, (\Psi_q'^l, \tau_q'^l) = \overline{\textsc{rights}}(\alpha_1,\ \Phi) \\ \mathtt{let}\ (\Psi_1^r, \tau_1^r), \ldots, (\Psi_r^r, \tau_r^r) = \textsc{rights}(\alpha_2,\ \Phi),\ (\varnothing, \alpha_2) \qquad \mathtt{let}\ (\Psi_1'^r, \tau_1'^r), \ldots, (\Psi_s'^r, \tau_s'^r) = \overline{\textsc{lefts}}(\alpha_2,\ \Phi) \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi_i^l \vee \Psi_j^r \vee \tau_i^l \leqslant \tau_j^r\ \}_{i=1}^{p}\ \}_{j=1}^{r} \rhd \Phi' \\ \Phi' \vdash \{\ \{\ \Psi \vee \Psi_i'^l \vee \Psi_j^r \vee \tau_j^r \not\leqslant \tau_i'^l\ \}_{i=1}^{q}\ \}_{j=1}^{r} \rhd \Phi'' \qquad \Phi'' \vdash \{\ \{\ \Psi \vee \Psi_i^l \vee \Psi_j'^r \vee \tau_j'^r \not\leqslant \tau_i^l\ \}_{i=1}^{p}\ \}_{j=1}^{s} \rhd \Phi'''\end{array}}{\Phi \vdash^{?} \Psi \vee \alpha_1 \leq \alpha_2 \rhd \Phi'''}$$

SVarNotLeqParamed

$$\frac{\begin{array}{c}\mathtt{let}\ (\Psi_1, \tau_1^r), \ldots, (\Psi_p, \tau_p^r) = \textsc{rights}(\alpha,\ \Phi) \\ \Phi \vdash \{\ \Psi \vee \Psi_i \vee \tau_i^r \not\leqslant (\alpha_1, \ldots, \alpha_n)\ \mathtt{t}\ \}_{i=1}^{p} \rhd \Phi'\end{array}}{\Phi \vdash^{?} \Psi \vee \alpha \not\leq (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \rhd \Phi'}$$

SVarNotLeqVar

$$\frac{\begin{array}{c}\mathtt{when}\ \alpha_1 \neq \alpha_2 \\ \mathtt{let}\ (\Psi_1^r, \tau_1^r), \ldots, (\Psi_p^r, \tau_p^r) = \textsc{rights}(\alpha_1,\ \Phi),\ (\varnothing, \alpha_1) \\ \mathtt{let}\ (\Psi_1^l, \tau_1^l), \ldots, (\Psi_q^l, \tau_q^l) = \textsc{lefts}(\alpha_2,\ \Phi),\ (\varnothing, \alpha_2) \\ \Phi \vdash \{\ \{\ \Psi \vee \Psi_i^r \vee \Psi_j^l \vee \tau_i^r \not\leqslant \tau_j^l\ \}_{i=1}^{p}\ \}_{j=1}^{q} \rhd \Phi'\end{array}}{\Phi \vdash^{?} \Psi \vee \alpha_1 \not\leq \alpha_2 \rhd \Phi'}$$

SParamedNotLeqVar

$$\frac{\mathtt{let}\ (\Psi_1, \tau_1^l), \ldots, (\Psi_p, \tau_p^l) = \textsc{lefts}(\alpha,\ \Phi) \qquad \Phi \vdash \{\ \Psi \vee \Psi_i \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leqslant \tau_i^l\ \}_{i=1}^{p} \rhd \Phi'}{\Phi \vdash^{?} \Psi \vee (\alpha_1, \ldots, \alpha_n)\ \mathtt{t} \not\leq \alpha \rhd \Phi'}$$